AI Assisted Coding

Lab Assignment 11.3

Name: K. Charan Yadav
Hall Ticket no: 2303A52367

Batch No: 20

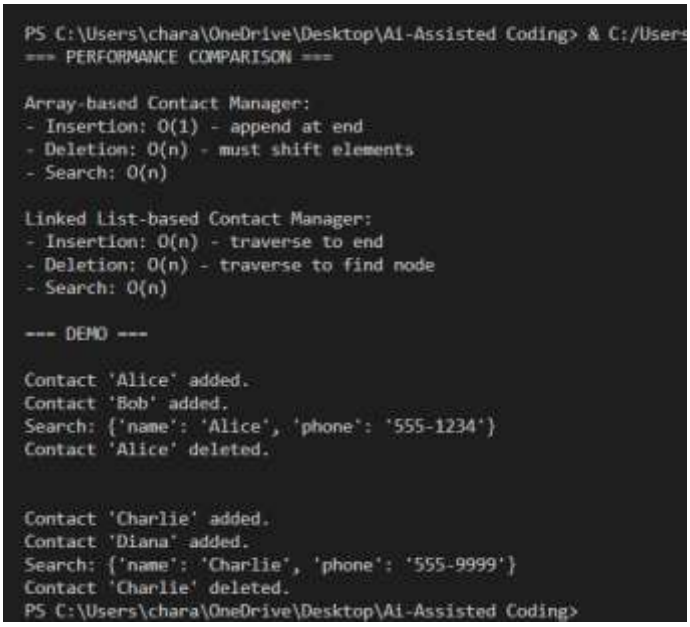
Task -1:

Prompt:

generate a Python Contact Manager using both array (list) and linked list data structures. The system should support adding, searching, deleting, and displaying contacts. I also requested case-insensitive search functionality and clear output messages. The goal was to compare static and dynamic data storage approaches.

```python

# Task- 1: Smart Contact Manager
# Array-based Contact Manager
class ArrayContactManager:
    def __init__(self):
        self.contacts = []

    def add_contact(self, name, phone):
        self.contacts.append({"name": name, "phone": phone})
        print(f"Contact '{name}' added.")

    def search_contact(self, name):
        for contact in self.contacts:
            if contact["name"].lower() == name.lower():
                return contact
        return None

    def delete_contact(self, name):
        for i, contact in enumerate(self.contacts):
            if contact["name"].lower() == name.lower():
                self.contacts.pop(i)
                print(f"Contact '{name}' deleted.")
                return True
        return False

    def display_contacts(self):
        if not self.contacts:
            print("No contacts found.")
        else:
            for contact in self.contacts:
                print(f"{contact['name']}: {contact['phone']}")


# Linked List Node
class ContactNode:
    def __init__(self, name, phone):
        self.data = {"name": name, "phone": phone}
        self.next = None

# Linked List-based Contact Manager
class LinkedListContactManager:
    def __init__(self):
        self.head = None
```

OUTPUT:



Justification:

Arrays are simple and allow fast insertion using append operations, making them suitable for small datasets. Linked lists provide dynamic memory allocation and efficient deletion without shifting elements. Both structures help understand different memory and performance behaviors. This comparison improves understanding of data structure efficiency.

Task 2:

Prompt: create a library request system using a FIFO queue and a priority queue in Python. The implementation needed enqueue and dequeue operations using deque and heapq. Faculty requests were required to have higher priority than student requests. The system should demonstrate request processing order.

```python
# Task 2: Library Book Search System (Queues & Priority Queues)

# Simple Queue for FIFO requests
class BookRequestQueue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, name, book_title, request_type):
        self.queue.append({"name": name, "book_title": book_title, "type": request_type})
        print(f"Request added: {name} ({request_type}) - {book_title}")

    def dequeue(self):
        if self.queue:
            request = self.queue.popleft()
            print(f"Processing: {request['name']} - {request['book_title']}")
            return request
        print("No requests in queue.")
        return None

    def display_queue(self):
        if not self.queue:
            print("Queue is empty.")
        else:
            for req in self.queue:
                print(f"  {req['name']} ({req['type']}) - {req['book_title']}")

# Priority Queue for Faculty-first requests
class BookRequestPriorityQueue:
    def __init__(self):
        self.heap = []
        self.counter = 0

    def enqueue(self, name, book_title, request_type):
        priority = 0 if request_type == "Faculty" else 1
        heapq.heappush(self.heap, (priority, self.counter, {"name": name, "book_title": book_title, "type": request_type}))
        self.counter += 1
        print(f"Request added: {name} ({request_type}) - {book_title}")

    def dequeue(self):
        if self.heap:
            priority, _, request = heapq.heappop(self.heap)
            print(f"Processing: {request['name']} ({request['type']}) - {request['book_title']}")
            return request
        print("No requests in queue.")
        return None
```

Output:

```
=== SIMPLE QUEUE (FIFO) ===
Request added: John (Student) - Python 101
Request added: Dr. Smith (Faculty) - Advanced Algorithms
Request added: Sarah (Student) - Data Science

Queue contents:
  John (Student) - Python 101
  Dr. Smith (Faculty) - Advanced Algorithms
  Sarah (Student) - Data Science

Processing requests:
Processing: John - Python 101
Processing: Dr. Smith - Advanced Algorithms

=== PRIORITY QUEUE (Faculty First) ===
Request added: John (Student) - Python 101
Request added: Dr. Smith (Faculty) - Advanced Algorithms
Request added: Sarah (Student) - Data Science
Request added: Prof. Johnson (Faculty) - Machine Learning

Queue contents:
  Dr. Smith (Faculty) - Advanced Algorithms
  Prof. Johnson (Faculty) - Machine Learning
  John (Student) - Python 101
  Sarah (Student) - Data Science

Processing requests (Faculty prioritized):
Processing: Dr. Smith (Faculty) - Advanced Algorithms
Processing: Prof. Johnson (Faculty) - Machine Learning
```

Justification:

A queue ensures fair processing of requests in arrival order using FIFO logic. A priority queue allows urgent requests, such as faculty requests, to be handled first. This combination models real-world service systems effectively. Both structures provide efficient O(1) or O(log n) operations.

Task-3:

Prompt:

To implement a stack-based ticket management system using Python. The stack needed push, pop, peek, and status-check operations. The implementation should simulate resolving tickets using LIFO behavior. The system also required a demonstration with multiple support tickets.

```
# Task - 3: Emergency Help Desk (Stack Implementation)
# Stack Node for tickets
class TicketNode:
    def __init__(self, ticket_id, student_name, issue):
        self.ticket_id = ticket_id
        self.student_name = student_name
        self.issue = issue
        self.next = None

# Stack-based Ticket Management System
class SupportTicketStack:
    def __init__(self, max_size=100):
        self.top = None
        self.size = 0
        self.max_size = max_size

    def push(self, ticket_id, student_name, issue):
        if self.is_full():
            print(f"Stack is full! Cannot add ticket {ticket_id}.")
            return False
        new_ticket = TicketNode(ticket_id, student_name, issue)
        new_ticket.next = self.top
        self.top = new_ticket
        self.size += 1
        print(f"Ticket #{ticket_id} added: {student_name} - {issue}")
        return True

    def pop(self):
        if self.is_empty():
            print("No tickets to resolve.")
            return None
        ticket = self.top
        self.top = self.top.next
        self.size -= 1
        print(f"Resolving Ticket #{ticket.ticket_id}: {ticket.student_name} - {ticket.issue}")
        return ticket

    def peek(self):
        if self.is_empty():
            print("No tickets in queue.")
            return None
        return self.top

    def is_empty(self):
        return self.size == 0
```

Output:

```
=== EMERGENCY HELP DESK (STACK) ===

Adding support tickets:

Ticket #101 added: Alice Johnson - Laptop won't connect to WiFi
Ticket #102 added: Bob Smith - Password reset failed
Ticket #103 added: Carol White - Software installation issue
Ticket #104 added: David Brown - Email not syncing
Ticket #105 added: Emma Davis - Printer driver problem

Current pending tickets:
Pending Tickets (LIFO order):
  1. Ticket #105: Emma Davis - Printer driver problem
  2. Ticket #104: David Brown - Email not syncing
  3. Ticket #103: Carol White - Software installation issue
  4. Ticket #102: Bob Smith - Password reset failed
  5. Ticket #101: Alice Johnson - Laptop won't connect to WiFi

Peek at next ticket to resolve:
  Next: Ticket #105

Resolving tickets (LIFO - Last In, First Out):

Resolving Ticket #105: Emma Davis - Printer driver problem
Resolving Ticket #104: David Brown - Email not syncing
Resolving Ticket #103: Carol White - Software installation issue
```

Justification:

A stack follows Last-In First-Out (LIFO), which matches emergency escalation handling. Recently raised issues are resolved first during urgent situations. Stack operations are efficient and simple to implement. This structure clearly demonstrates real-world problem prioritization.

Task-4:

Prompt:

Generate a hash table implementation in Python with insert, search, and delete operations. The solution needed collision handling using chaining. The code should include comments explaining hashing and indexing. The goal was to understand fast data retrieval techniques.

```python
# Task - 4: # Hash Table with Chaining for Collision Handling
class HashTable:
    def __init__(self, size=10):
        #Initialize hash table with given size
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        #Generate hash value for key using simple modulo function
        return hash(key) % self.size

    def insert(self, key, value):
        #Insert key-value pair into hash table
        hash_index = self._hash(key)

        # Check if key already exists and update it
        for i, (k, v) in enumerate(self.table[hash_index]):
            if k == key:
                self.table[hash_index][i] = (key, value)
                print(f"Updated: {key} -> {value}")
                return

        # Add new key-value pair (chaining handles collision)
        self.table[hash_index].append((key, value))
        print(f"Inserted: {key} -> {value}")

    def search(self, key):
        #Search for value by key
        hash_index = self._hash(key)

        # Linear search through chain at hash index
        for k, v in self.table[hash_index]:
            if k == key:
                print(f"Found: {key} -> {v}")
                return v

        print(f"Not found: {key}")
        return None

    def delete(self, key):
        #Delete key-value pair from hash table
        hash_index = self._hash(key)

        # Search and remove from chain
        for i, (k, v) in enumerate(self.table[hash_index]):
```

Output:

```
=== HASH TABLE WITH CHAINING ===

Inserting entries:

Inserted: name -> Alice
Inserted: age -> 25
Inserted: city -> NYC
Inserted: job -> Engineer
Updated: age -> 26

Searching for entries:

Found: name -> Alice
Found: age -> 26
Not found: country

Hash table structure:
Hash Table Contents:
  Index 0: [('age', 26)]
  Index 3: [('name', 'Alice'), ('city', 'NYC'), ('job', 'Engineer')]

Deleting entries:

Deleted: city
Key not found: country

Final hash table:
Hash Table Contents:
  Index 0: [('age', 26)]
  Index 3: [('name', 'Alice'), ('job', 'Engineer')]
PS C:\Users\chara\OneDrive\Desktop\AI-Assisted Coding>
```

Justification:

Hash tables allow very fast data access using key-based indexing. Collision handling through chaining prevents data loss when keys map to the same index. This structure provides average O(1) search performance. It is widely used in databases and real-time lookup systems.

Task-5:

Prompt:

design a campus management system using suitable data structures for different features such as attendance, event registration, library borrowing, bus scheduling, and cafeteria orders. One feature needed full implementation. The solution also required justification for each structure selection.

```
# Task - 5:
# Real-Time Campus Resource Management System

# Feature 1: Student Attendance Tracking - Using Dictionary (Hash Map)
# Justification: O(1) average lookup and insertion for student records.
# Perfect for quick attendance marking and retrieval by student ID.

class AttendanceSystem:
    def __init__(self):
        self.attendance = {}

    def mark_attendance(self, student_id, date, status):
        if student_id not in self.attendance:
            self.attendance[student_id] = {}
        self.attendance[student_id][date] = status
        print(f"Marked {student_id} as {status} on {date}")

    def get_attendance(self, student_id):
        if student_id in self.attendance:
            print(f"Attendance for {student_id}:")
            for date, status in self.attendance[student_id].items():
                print(f"  {date}: {status}")
            return self.attendance[student_id]
        print(f"No records for {student_id}")
        return None

    def get_attendance_percentage(self, student_id):
        if student_id not in self.attendance:
            return 0
        records = self.attendance[student_id]
        present = sum(1 for s in records.values() if s == "Present")
        percentage = (present / len(records)) * 100 if records else 0
        print(f"{student_id} attendance: {percentage:.1f}%")
        return percentage

# Feature 2: Event Registration System - Using Set
# Justification: Sets provide O(1) lookup and prevent duplicate registrations.
# No duplicates allowed; perfect for managing unique student registrations.

class EventRegistration:
    def __init__(self, event_name):
        self.event_name = event_name
        self.registered_students = set()

    def register(self, student_id):
```

Output:

```
--- CAFETERIA ORDER QUEUE (Selected Feature) ---

Order #1 placed for S001: ['Pizza', 'Coke']
Order #2 placed for S002: ['Burger', 'Fries']
Order #3 placed for S003: ['Salad', 'Water']

Pending orders:
Pending Orders:
  #1: S001 - ['Pizza', 'Coke']
  #2: S002 - ['Burger', 'Fries']
  #3: S003 - ['Salad', 'Water']

Processing orders:
Order #1 completed for S001
Order #2 completed for S002

Remaining orders:
Pending Orders:
  #3: S003 - ['Salad', 'Water']

--- OTHER FEATURES DEMO ---

Marked S001 as Present on 2024-01-15
Marked S001 as Absent on 2024-01-16
S001 attendance: 50.0%
S001 registered for Tech Conference
S002 registered for Tech Conference
Tech Conference: 2 students registered
S001 borrowed 'Python Guide' (Due: 2024-02-01)
Books borrowed by S001:
  - Python Guide (Due: 2024-02-01)
Route B1 added: 08:00
All Bus Routes:
  B1: ['Gate', 'Hall A', 'Library'] - Departs: 08:00
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
```

Justification:

Different system components require different data structures for efficiency. Dictionaries provide fast attendance lookup, sets prevent duplicate registrations, and queues manage ordered requests. Selecting appropriate structures improves performance and scalability. This demonstrates practical application of data structures in real systems.